Python Comprehensions

Table of contents

List comprehension 1
Dictionary comprehension 2
Nested comprehensions 2
Filtering 2
The general pattern 3
`True`/`False` as `1`/`0` 3
`all` and `any` 4

A **comprehension** is a concise way to build a list or dictionary directly from an iterable, without writing an explicit accumulator for-loop.

List comprehension

```
>>> [k**2 for k in range(4)]
[0, 1, 4, 9]
```

Equivalent to:

```
>>> acc = []
>>> for k in range(4):
...     acc.append(k**2)
```

`sum(... for ...)`, without the surrounding `[]`, computes a running total directly. The bracket-less form is a *generator expression*, and so totals without building the intermediate list.

```
>>> sum(k**2 for k in range(4))
14
```

Dictionary comprehension

Same shape, but the output expression is a `key: value` pair and the braces are `{}`:

```
>>> {x: f"value is {x}" for x in range(3)}
{0: 'value is 0', 1: 'value is 1', 2: 'value is 2'}
```

Useful for assigning default values across a set of keys:

```
>>> {x: 0 for x in "ABC"}
{'A': 0, 'B': 0, 'C': 0}
```

Duplicate keys collapse — iterating a string with repeated characters still yields one entry per distinct character:

```
>>> cs = "hello world"
>>> {x: cs.count(x) for x in cs}
{'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

Nested comprehensions

Multiple `for` clauses can appear in one comprehension. The order of the `for` clauses matters — it matches the order they'd be nested as for-loops:

```
>>> [(a, x) for a in "AB" for x in range(2)]
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]

>>> [(a, x) for x in range(2) for a in "AB"]
[('A', 0), ('B', 0), ('A', 1), ('B', 1)]
```

Filtering

A trailing `if` filters which elements are included:

```
>>> [x for x in range(101) if not (x % 3) and not (x % 7)]
[0, 21, 42, 63, 84]
```

The general pattern

```
[x for x in xs if P(x)]
[x for x in xs for y in ys if P(x, y)]
[x for x in xs for y in ys for z in zs if P(x, y, z)]
...
```

where `P` is some predicate.

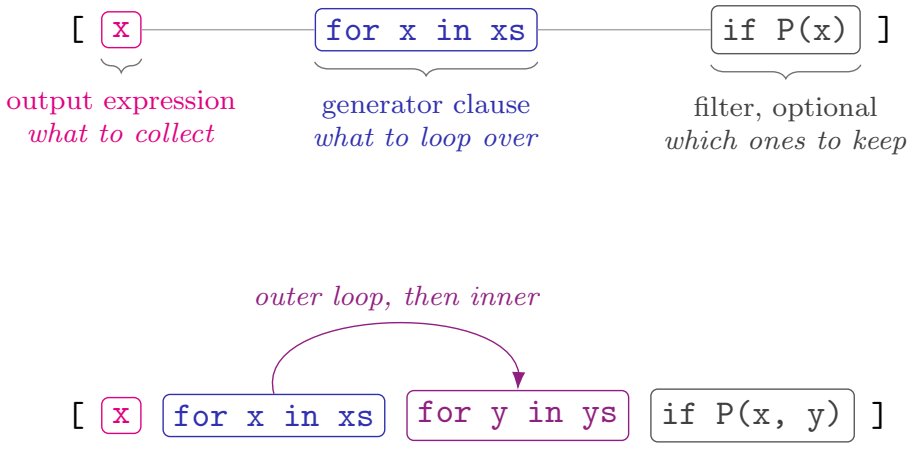

Figure 1: The three parts of a comprehension, named on the first two forms of the general pattern: output expression, generator clause and filter. Magenta is the output expression, blue the first generator clause, purple the second. That key covers the boxed code only: the purple arrow labelled *outer loop, then inner* and the purple line beneath the second form are annotations, not a fourth part.

The same three parts appear in a dictionary comprehension, except the output expression is a `key: value` pair.

`True`/`False` as `1`/`0`

`True`/`False` behave as `1`/`0` in arithmetic — a common idiom for counting how many elements of an iterable satisfy some condition. It works because Python's `bool` is a subtype of `int`.

```
>>> True + True + False
2
```

```
>>> sum(x > 0 for x in [-1, 2, 3, -4])
2
```

`all` and `any`

Python's built-in `all(xs)`/`any(xs)` check whether every/some element of `xs` is truthy:

```
>>> all(['A' <= x <= 'Z' for x in "HELLO WORLD"])
False
>>> any(['A' <= x <= 'Z' for x in "HELLO WORLD"])
True
```

> Careful: redefining a function named `all` or `any` in your own code shadows these built-ins for the rest of that scope. 2025-08-25-comprehensions[1]'s exercises do exactly that, deliberately, to reimplement them.

See python-for-loops[2] for the equivalent for-loop/accumulator forms these shortcut.

[1] courses/csse1001/2025-08-25-comprehensions.pdf
[2] courses/csse1001/python-for-loops.pdf